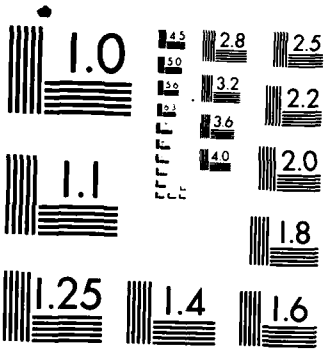
AD-A161 368 FINDING CRITICAL SETS(U) DUKE UNIV DURHAM NC DEPT OF COMPUTER SCIENCE D W LOVELAND 13 SEP 85 CS-1985-22 AFOSR-TR-85-0951 AFOSR-83-0205 1/1

UNCLASSIFIED F/G 12/1 NL



(2)

CS-1985-22

Finding Critical Sets

Donald W. Loveland
Duke University

DEPARTMENT OF COMPUTER SCIENCE

DUKE UNIVERSITY

Uncl

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR- 85-0911	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Finding Critical Sets	5. TYPE OF REPORT & PERIOD COVERED: Technical paper
	6. PERFORMING ORG. REPORT NUMBER: CS-1985-22
7. AUTHOR(s): D.W. Loveland	8. CONTRACT OR GRANT NUMBER(s): AFOSR 83-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Department, Duke University, Durham, NC 27706	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F, 2304/A7
11. CONTROLLING OFFICE NAME AND ADDRESS: same as II 14	12. REPORT DATE: September 13, 1985
	13. NUMBER OF PAGES: 13
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Office of Scientific Research, Air Force System Command, Bolling AFB; Washington, DC 20332	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release,
distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Submitted for publication

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Algorithms, monotone set functions, performance bounds

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is a major revision of a paper by the same title. Several algorithms are given for finding a critical subset S determined by a binary monotonic set function f. A set S is critical iff f(T) = 1 for all T such that $S \subseteq T$ and f(h) = 0 for all R such that $R \subset S$. Although the first critical set can be found very quickly, finding additional critical sets can become very difficult.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Uncl

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

FINDING CRITICAL SETS[1]

Donald W. Loveland
Duke University

This report is a major revision of report CS-1982-23

[1]This research has been partially supported by grants AFOSR 81-0221 and AFOSR 83-0205 under the sponsorship of the Air Force Office of Scientific Research, Air Force Systems Command.

ABSTRACT

Several algorithms are given for finding a critical subset S determined by a binary monotonic set function over a given set U. A set S is critical iff $f(T) = 1$ for all T such that $S \subseteq T$ and $f(R) = 0$ for all R such that $R \subset S$. Upper bounds on the worst case times for the algorithms, determined by the number of calls to the binary set function, vary downwards from $2r\lceil lg_2\, n\rceil$ function calls. Here n is the size of U and r is the size of the critical set found. We find that the conceptually easiest algorithm is not the easiest to program and also that the algorithms with better upper bounds fare more poorly in practice, based on a small sample of random trials. Although finding one critical set is easy we also show that it can quickly get very difficult to find additional critical sets, simply because it can be hard to separate known critical sets from further possible critical sets.

1. Introduction

By a *binary monotone* set function we mean a function defined on the power set of set U such that if $f(S) = 1$ then $f(S_1) = 1$ for all S_1 such that $S \subseteq S_1$, and takes only 0 and 1 as values. A *critical set* S *of* f satisfies $f(S) = 1$ and $f(S_2) = 0$ for all S_2 such that $S_2 \subset S$. (We use $\subset$ for "$\subseteq$ and $\neq$".) We give algorithms for finding a critical set of f (often called simply "a critical set" when f is understood); using at most $2r\lceil lg\ n \rceil$ function calls, where r is the number of elements of the critical set, n is the number of elements in the given domain ("universe"), $\lceil x \rceil$ is the least integer m satisfying $m \geq x$, and $lg\ n$ denotes $\log_2 n$. After presentation of the algorithms we consider implementation of the algorithms since here the manner of implementation affects not only the running time of the algorithm but the relative ease of implementation as well. Finally, we will see in the last section that finding another critical set can get very difficult quickly (exponential growth) as a function of the number of known critical sets.

The problem of finding critical sets arises, for example, in classification networks with a monotone property. Given a set of input nodes and a set of output nodes for a graph, suppose firing certain subsets of input nodes are supposed to result in one output per subset being fired. Also, any superset of a valid input fires the same output node. Clearly, firing all input nodes simultaneously usually leads to many output nodes being fired. Consider minimal input sets that fire two or more output nodes. We have interest in locating these minimally ambiguous nodes. See [1] for a paper that reports on this application, and its role in constructing "expert knowledge" systems. Also, a use for a generalization of the problem to non-boolean distributive lattices has arisen in devising tests for input attenuation values in networks abstracting certain expert system inference systems using uncertainty.

2. The algorithms.

We now give four different algorithms to solve the critical set problem, each of independent interest The first critical set algorithm we give is particularly simple conceptually and, on occasion, can perform better than the other algorithms. For the remainder of the paper we will assume that there exists at least

one critical set for the monotone set function f. Note that this is determined by $f(U) = 1$ and $f(\emptyset) = 0$, where U is the universal set. We represent the cardinality of set S by $|S|$.

Algorithm I

{ Assumption: a critical set exists }

1. $k \leftarrow 1$

   $U^* \leftarrow U$

2. Partition U^* into roughly equal subsets $U_1, U_2, \ldots, U_{2k}$

   $U^{**} \leftarrow U^*$

3. For $i \leftarrow 1$ to $2k$ do

   if $f(U^* - U_i) = 1$ then $U^* \leftarrow U^* - U_i$

4. If $\max |U_i| = 1$ then U^* is a critical set

   else if $|U^{**}| \geq 2|U^*|$ then return to step 2

   else $k \leftarrow \min(2k, |U^*|)$ and return to step 2.

The phrase "roughly equal" means that the two sets differ at most by one in cardinality.

The underlying concept is best seen when one critical set C of r elements exists. If $k = r$ then dividing U^* into $2r$ equal disjoint subsets allows the r elements of C to be in only half the subsets. For each U_i containing no element of C, $f(U^* - U_i) = 1$ and U_i may be removed from U^*. With many critical sets present, the same principle works; the algorithm "focusses in" on one critical set in the sense that critical sets with elements in U_i are discarded whenever $U - U_i$ contains a critical set, since then $f(U - U_i) = 1$. The evaluation with all U_i equal to 1 is needed to confirm that one actually has a critical set.

We now consider an upper bound on the number of function calls needed by this algorithm to locate a critical set. The number of function calls is the appropriate measure when the function evaluation is non-

trivial because the rest of the algorithm processes very quickly. If the elements of the critical set of size r that we finally locate are "equidistant" from each other, then until k reaches size r the set variable U' never shrinks. (We assume that r and n are powers of 2 for convenience.) Thus $\sum_{i=0}^{\log r} 2*2^i \leq 4\log r$ function calls are made before the *if* condition in Step 3 is met for the first time. Thereafter, with $k=r$ throughout the remainder of the run, we have $2r(\log n - \log r)$ function calls in total as the U' shrinks by a factor of 2 for each execution of Step 3. Combining these two parts we have

$$4\log r + 2r\log n - 2r\log r$$

$$\leq 2r\log n$$

function calls needed for Algorithm I.

The second algorithm takes the depth-first approach which allows us to set tighter upper bounds conveniently. Here we repeatedly find one element of the critical set at a time, and doing this by continual binary splitting of the space under consideration. Three set variables are important: C retains known members of the critical set being isolated, A includes at least one element of the critical set being isolated, and R includes the remaining unknown members of that critical set. Correctness follows from the fact that whenever A is non-empty then A must contain members of the critical set being isolated.

Algorithm II

{ Assumption: a critical set exists }

1. $C \leftarrow \emptyset$

   $A \leftarrow U$

   $R \leftarrow \emptyset$

2. Split A into roughly equal disjoint sets A_1 and A_2 such that $|A_1| \leq |A_2|$

3. If $f(C \cup R \cup A_1) = 1$ then $A \leftarrow A_1$

   else { A_2 contains elements of the critical set being isolated }

   $R \leftarrow A_1 \cup R$

   $A \leftarrow A_2$

4. If $|A| > 1$ then return to step 2

else { an element of the critical set is isolated }

$C \leftarrow A \cup C$

$A \leftarrow R$

$R \leftarrow \emptyset$

if $A = \emptyset$ then C is a critical set

else return to step 2

The inequality in Step 2 assures the proper handling of the case $|A| = 1$. It is important but straightforward to notice that C is a critical set at termination without checking C explicitly.

An upper bound of $r \log n$ function calls, where r is the number of elements in the critical set found, follows directly from the depth-first nature of the algorithm.

We are able to construct a variant of Algorithm II that gives a better upper bound for large r, that is, that seems better when the critical set found is large. We note that these algorithms *tend* to find small critical sets so that most critical sets of a function must be large to have the critical set found be large. However, since the bound is as good uniformly as that for Algorithm I and very good when large critical sets are found, which is when computational efficiency really matters, this algorithm definitely is interesting.

This algorithm is seen to differ from the previous algorithm primarily in the first portion where we "grow" an interval first.

Algorithm III

{ Assumption: a critical set exists }

1. $R \leftarrow U$

   $C \leftarrow \emptyset$

2. $S \leftarrow$ some subset of R of size 2

   $i \leftarrow 1$

3. While $f(C \cup R - S) = 1$ do

   $R \leftarrow R - S$

   $S \leftarrow$ some subset of R of size 2^i

   $i \leftarrow i + 1$

4. $A \leftarrow S$

   $R \leftarrow R - S$

5. Split A into roughly equal disjoint sets A_1 and A_2 such that $|A_1| \leq |A_2|$

6. If $f(C \cup R \cup A_1) = 1$ then $A \leftarrow A_1$

   else { A_2 contains elements of the critical set being isolated }

   $R \leftarrow A_1 \cup R$

   $A \leftarrow A_2$

7. If $|A| > 1$ then return to step 5

   else { an element of the critical set is isolated }

   $C \leftarrow A \cup C$

   $A \leftarrow R$

   $R \leftarrow \emptyset$

   if $A = \emptyset$ then C is a critical set

   else return to step 5

To compute a bound on the number of function calls, we consider the effect of pruning the space with the sets of increasing size, as done in steps 1-4. Let s_k be the size of the set which cannot be discarded in the attempt to find the k th element of the critical set (so s_k is the size of the set assigned to variable A at step 4), and let r be the number of elements of the isolated critical set. Then there are $\log s_k$ function calls during the discovery of the set of size s_k and the same number of calls to isolate the critical element in that set. Thus we can bound the number of calls by

$$\sum_{j=1}^{r} 2\log s_j .$$

If we represent our universe as a binary vector and choose the elements of our critical set at positions $i*(n/r)$ for $i=1,\ldots,r$, which is the worst case critical set of r elements, then $s_i = n/r$ and the number of function calls is

$$\sum^{r} 2\log n/r = 2r \log n/r .$$

If $r = cn$, for $c<1$, then we have

$$2cn \log 1/c = Cn, \text{ for some } C .$$

In particular, if $r = n/2$ because all critical sets are that size or greater, then the number of function calls is bounded by n, a nice consequence because it is clear intuitively that when r is close to n (and this is known) then the best thing to do is to test $f(U-e)$ for each element e of the universe, performing $U \leftarrow U-e$ whenever the function evaluation is 1, to obtain the critical set directly. This, of course, takes n function calls.

The fourth algorithm is basically a depth-first algorithm like Algorithm II, with the primary difference the introduction of a stack to retain sets known to contain points. The stack provides much more structure to the residual set R, often shortening the search for the next critical set element. The set R is defined to be the union of sets on the stack. Although the stack device works equally well on Algorithm II, the symmetric check of both halves of the split is more natural when the stack is used.

Algorithm IV

{ Assumption: a critical set exists }

1. $C \leftarrow \emptyset$

   $A \leftarrow U$

   empty_stack_R $\leftarrow$ true

2. Split A into roughly equal disjoint sets A_1 and A_2

3. If $f(C \cup R \cup A_1) = 1$ then $A \leftarrow A_1$

   else

   if $f(C \cup R \cup A_2) = 1$ then $A \leftarrow A_2$

   else { both A_1 and A_2 contain elements of the critical set being isolated }

   push_R (A_1)

   $A \leftarrow A_2$

4. If $|A| > 1$ then return to step 2

   else { an element of the critical set is isolated }

   $C \leftarrow A \cup C$

   if empty_stack_R then C is a critical set

   else pop_R(A) and return to step 4.

It is again important but straightforward to notice that C is a critical set at termination without checking C explicitly. Here the manner of splitting A when $|A|$ is odd is unimportant.

The number of function calls used here is bounded by $2r \log n$, an easy upper bound because there are r elements to our isolated critical set, each using at most $2\log n$ function calls to be isolated. Although this bound is approachable for very small r, (e.g. $r=1$), in general the use of a stack provides much better performance, as we amplify in the next section.

3. Remarks on implementations.

The four above algorithms, and several variants thereof, were implemented to observe relative performance qualitatively, to allow some idea of performance relative to the upper bounds determined above, and to see if there were any lessons to be learned regarding implementation. There were interesting observations in all these regards.

The first attempt to implement the algorithms was undertaken in PASCAL (as were all subsequent implementations) with use of the PASCAL set type, a data type allowing the manipulation of sets directly as primitives, with the set algebra operations and the membership relation. The first programs were obtained using a direct translation of the algorithms into PASCAL using this data type. Some trial runs, over a space of 2048 elements and a random mix of binary functions, revealed a worse time than predicted and with Algorithm I far superior to Algorithm IV, the only other algorithm first programmed. The problem was immediately discernible. For this implementation approach the partition step (step 2 in both Algorithms I and IV) takes $O(n)$ time because the only general way to partition a set is to test membership using the membership relation, and to sweep the entire universe. (Again, n is the universe size.) While generally costly, it was much less costly for Algorithm I than for the others because it is as easy to split into $2k$ sets as to split into two sets. An order-of-magnitude argument shows this nicely, as we now show.

We consider the running time of Algorithms I and IV under the assumption that computation of the binary set function is fast, indeed a unit operation (perhaps not realistic in most cases, but nearly true in our test runs). Roughly, Algorithm I partitions a set $O(log\ n)$ times, the partition costing $O(n)$ time and step 3 costing $O(r)$ time (at most). Thus the cost is bounded by $O((n+r)\log n)$ effort. For Algorithm IV the number of partitions is roughly the number of function calls and thus the total effort is closer to $O(nr \log n)$. Algorithms II and III clearly behave as Algorithm IV does.

However, there is another approach to implementing Algorithm IV, actually the implementation intended when Algorithm IV was designed, that turned out to be beneficial to all algorithms. The idea is to use intervals to define the primitive sets, i.e. to consider elements as numbered from 1 to n and use

ordered pairs of numbers to specify end points of all numbers in a primitive set. In fact, we learned that PASCAL permits the construct $[a..b]$ to denote the "interval set" defined by variables a and b, thus permitting a calculus of intervals within the PASCAL set type domain. The beauty of the interval notation is that partitioning a set into two "equal" parts is a matter of computing a midpoint of an interval, a unit operation. Moreover, in Algorithm IV the intervals are adjacent in R so that R (the union of sets on the stack) remains a single interval throughout the entire computation. Algorithm I gained some speed by this move but because any partition set could be eliminated, the universe quickly became complex and partitioning required a non-trivial program, although $O(r)$ rather than $O(n)$ steps are needed.

We take our actual running times with a barrel of brine because there is no manner in which to characterize "typical" monotone binary set functions, or to envision likely application structures. (In the application that led to the investigation of this problem there seems to be few critical sets when there exist any.) We used a space of 512 elements with binary functions that defined up to 16 critical sets generated randomly regarding size and location, under obvious constraints. The critical sets were both specified by randomly chosen points or randomly chosen intervals, with cardinality from 1 to 450, although ranges were usually restricted to a narrower band than that. What we are willing to note is that all algorithms ran well below their computed bounds in number of function calls, and that Algorithms I and IV almost uniformly beat the other algorithms, many times by a factor of two or more, but many times the difference was less. Recall that Algorithm II does not incorporate a stack for R; with such a stack it would perform comparably to Algorithm IV. Algorithm IV was almost always the winner but usually by 10% or less over Algorithm I. Examples exist where each of these two algorithms severely beat the other. All algorithms ran quickly so that the computation time would be in the evaluation of the function in most applications.

We judge Algorithm IV the winner, primarily on the ease of programming the interval version discussed above, but also because it slightly outperformed the next best algorithm on the average. The use of stacks to retain partially selected sets clearly aids efficiency in depth-first implementations. The primary surprise at first was that a combination of Algorithm IV with the front-end of algorithm III (that eliminated ever-increasing intervals) performed less well than either of Algorithm I or IV, although

performance was comparable when all critical sets were large. The reason seems to be that the symmetric check is largely wasted when the intervals are preprocessed for relevance, which is what the front-end accomplishes.

It is always interesting to note when examples with better analytic upper bounds fare less well than algorithms with poorer analytic upper bounds. Algorithm III looks best from upper bound considerations alone but generally faired poorly in comparison with two other algorithms.

4. Finding many critical sets.

The critical set algorithms find only one critical set although in general a monotone function defines many critical sets. How are (the) others found? It is clear that a second critical set can be found, if one exists, by taking each element c of the first critical set in turn and evaluating $f(U - \{c\})$. If the value 1 is returned for any such c then a new critical set exists as a subset of $U - \{c\}$. Then $U - \{c\}$ is to be used as the new universe to find a new critical set.

In general, suppose k critical sets are already discovered and we wish to know if there is a $k+1^{st}$ and, if so, to be given a set containing the $k+1^{st}$ critical set (a containment set) not containing any of the k known critical sets. Let S_i denote the i^{th} known critical set with $S_i = \{c_{i1},\ldots,c_{ir_i}\}$, and suppose all these sets are pairwise disjoint. Then it suffices to consider the sets $C_\alpha = \{c_{1j_1}, c_{2j_2},\ldots,c_{kj_k}\}$, where $1 \leq j_i \leq r_i$. (Here $\alpha = j_1 j_2 \cdots j_k$.) If each $r_i = k$ then there could be k^k C_α's to consider. That is, $f(U - C_\alpha)$ might need to be evaluated for all possible C_α. Consider the following. Suppose $U - \{c_{1j_1}, c_{2j_2},\ldots, c_{kj_k}\}(= U - C_{\alpha_0})$ is a critical set for some fixed integers $j_1, j_2,\ldots, j_k$ satisfying $j_i \leq k = r_i$. If the S_i, $1 \leq i \leq k$, are all found first (a good possibility) then it could take $k^k - 1$ function calls of form $f(U - C_\alpha)$, where each returns 0, before $f(U - C_{\alpha_0})$ is evaluated with value 1 returned. Thus, the problem of finding a set that contains only new critical sets can be extremely costly even when only one more critical set exists. We emphasize that the complexity of finding a critical set is a function of the number of critical sets already found; the first critical set is not hard to find.

We close with a further observation. It might be possible to devise another type of algorithm that

would yield all critical sets quickly, without having to find new containment sets. However, an argument based on the same principle as above shows that, if function calls are the only source of information used to find critical sets, then finding critical sets has to be hard. For an n element universe $\Omega(2^n/n)$ function calls can be needed just to determine the number of critical sets, or to determine the critical sets in a case when the description of the class of critical sets is very straightforward. Let a k-set mean a set with k elements. Then the description of a critical set class hard to confirm is "the class of all n/2-sets".

We outline the argument for the above. Let U have n elements. Let H denote the class of all n/2-sets and for each $h \in H$ consider a monotonic function f_h on U such that every set in $H - \{h\}$ is a critical set but h is not. I.e., $f_h(h) = 0$. There are $\Omega(2^n/n)$ such f_h since $|H| = \Omega(2^n/n)$. But we note that for all $S \subseteq U$ such that $|S| \geq n/2 + 1$ we have $f_h(S) = 1$ because S contains an n/2-set different than h. Also, $f_h(S) = 0$ for all $S \subseteq U$ such that $|S| \leq n/2 - 1$. These hold independently of the choice of h. Thus, only function calls for n/2-sets can yield information and there is no dependency between n/2-sets regarding f values. Therefore at least $2^n/n$ function evaluations may be needed to determine if some n/2-set is not a critical set or if indeed all n/2-sets are critical sets. In the latter case there is one more critical set than otherwise. This simultaneously illustrates that the determination of the fact "all n/2-sets are critical sets" is hard and determination of the number of critical sets is hard. This is what we wished to show.

5. Acknowledgments

We wish to thank Paul Lanzkron for his expertise in and patience in implementing numerous versions of the algorithms that the author devised. We also thank the referee for contributing the very interesting Algorithm I and thank Ian Munro for the suggestion of eliminating ever-increasing intervals that led to Algorithm III.

References:

1. D. W. Loveland and M. Valtorta. Detecting ambiguity: an example in knowledge evaluation. *Proc. of the Eighth Intern. Joint Conf. on Artif. Intell.*, Karlsruhe, Aug. 1983, pp. 182-184.

END

FILMED

1-86

DTIC